

AD-A019 706

SEMANOL (73) INTERPRETER DOCUMENTATION

Paul T. Berning

TRW Systems Group

Prepared for:

Rome Air Development Center

30 June 1975

DISTRIBUTED BY:

NTIS

National Technical Information Service
U. S. DEPARTMENT OF COMMERCE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER RADG-TR-75-211, Vol IV (of four)	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) SEMANOL (73) INTERPRETER DOCUMENTATION	5. TYPE OF REPORT & PERIOD COVERED Final Technical Report March 1974 - March 1975	
7. AUTHOR(s) Paul T. Berning	6. PERFORMING ORG. REPORT NUMBER N/A	
9. PERFORMING ORGANIZATION NAME AND ADDRESS TRW Systems Group One Space Park Redondo Beach CA 90278	8. CONTRACT OR GRANT NUMBER(s) F30602-74-C-0067	
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (ISIS) Griffiss AFB NY 13441	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 63728F 55500804	
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Same	12. REPORT DATE 30 June 1975	
	13. NUMBER OF PAGES 22 30	
	15. SECURITY CLASS. (of this report) UNCLASSIFIED	
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report) Same		
18. SUPPLEMENTARY NOTES RADG Project Engineer: Captain John M. Ives/ISIS		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Metalanguage, JOVIAL, JOVIAL (J73), SEMANOL, SEMANOL (73), compiler, language standardization, interpreter, language definition, language control, syntax, semantics, language grammar, SIL, CMS-2, Command and Control language, Parse Tree, software reliability		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) The formal definition of the programming language JOVIAL (J73) was produced by the metalanguage, SEMANOL. The process of definition resulted in the metalanguage, SEMANOL. The process of definition resulted in the successful identification of many ambiguities and conflicts, in the JOVIAL language which were reported to the language definition committee. SEMANOL (73) is under- standable by laymen and processable by the SEMANOL interpreter computer program. The interpreter program was completed and debugged during the contract period. (Cont'd)		

DD FORM 1473
1 JAN 73

EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

JOVIAL (J73), as processed by the SEMANOL interpreter, has been tested to the extent that (1) the JOVIAL (J73) level one subset grammar is well debugged, (2) the formal definition is syntactically correct and (3) the simpler semantics are tested to yield correct answers. The results of this effort coupled with previous and concurrent efforts show SEMANOL as a highly valuable standardization tool for failure use in DOD language controls.

ADD SIGNATURE

DATE

INITIALS

DATE

BY

DATE

ii

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

DC
029071

RADC-TR-75-211, Vol IV (of four)
Final Technical Report
30 June 1975

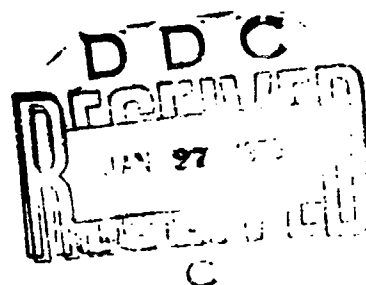


SEMANOL (73) INTERPRETER DOCUMENTATION

TRW Systems Group

ADA019706

Approved for public release;
distribution unlimited.

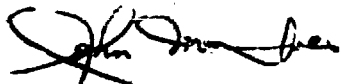


Rome Air Development Center
Air Force Systems Command
Griffiss Air Force Base, New York 13441

This report has been reviewed by the RADC Information Office (OI) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public including foreign nations.

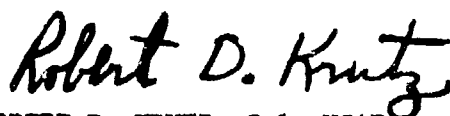
This report has been reviewed and is approved for publication.

APPROVED:



JOHN M. IVES, Capt, USAF
Project Engineer

APPROVED:



ROBERT D. KRUTZ, Col, USAF
Chief, Information Sciences Division

FOR THE COMMANDER:



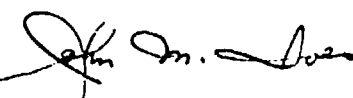
JOHN P. HUSS
Acting Chief, Plans Office

Do not return this copy. Retain or destroy.

EVALUATION

F30602-74-C-0067, SEMANOL (73)

The formal definition of the programming language JOVIAL (J73) was produced by the metalanguage, SEMANOL (73). The process of definition resulted in the identification of many ambiguities and conflicts in the J73 language under development. At the same time SEMANOL (73) is understandable by laymen and likewise processable by the SEMANOL interpreter computer program, making it highly amenable to locating and debugging potential compiler errors or uncertainties. This value is of enough significance that SEMANOL (73) will be implemented as an integral key requirement tool for use in the forthcoming USAF Higher Order Language Control Facility.



JOHN M. IVES, Capt, USAF
Project Engineer
Software Sciences Section

PREFACE

This document has been prepared for Rome Air Development Center in accord with CDRL Sequence Number A004 and paragraph 4.1.5 of the Statement of Work of contract number F30602-74-C-0067. It is the program documentation of the SEMANOL(73) Interpreter. This program documentation describes the internal organization of the SEMANOL(73) Interpreter and provides instructions on the use of the program. Note that the source program listings themselves contain detailed documentation about internal program operation in the form of comments. These comments constitute an extension to the material in this report.

CONTENTS

1. Program Name.	1
2. Abstract.	1
3. Machine Definition.	1
4. Program Description	1
4.1 The Translator.	3
4.2 The Interpreter	7
4.3 Tnames.	18
5. Logic Diagrams.	18
6. Inputs.	18
6.1 Translator Input.	18
6.2 Interpreter Input	18
6.3 Tnames Input.	19
7. Output.	20
7.1 Translator Outputs.	20
7.2 Interpreter Outputs	20
7.3 Tnames Output	20
8. Program Setup	21
9. Operator Instructions	22

1. Program Name

The name of this total computer program is the "SEMANOL(73) Interpreter". The SEMANOL(73) Interpreter is then made up of two major subprograms called the "Translator" and the "Interpreter". There is also a much smaller auxiliary program called "Tnames".

2. Abstract

The SEMANOL(73) Interpreter takes a SEMANOL(73) meta-language program describing a computer programming language and uses that meta-language description to control its interpretive execution of a program given to it in the programming language described. It is thus a generalized interpreter of higher order language (or machine language if desired) which is driven by a SEMANOL(73) description.

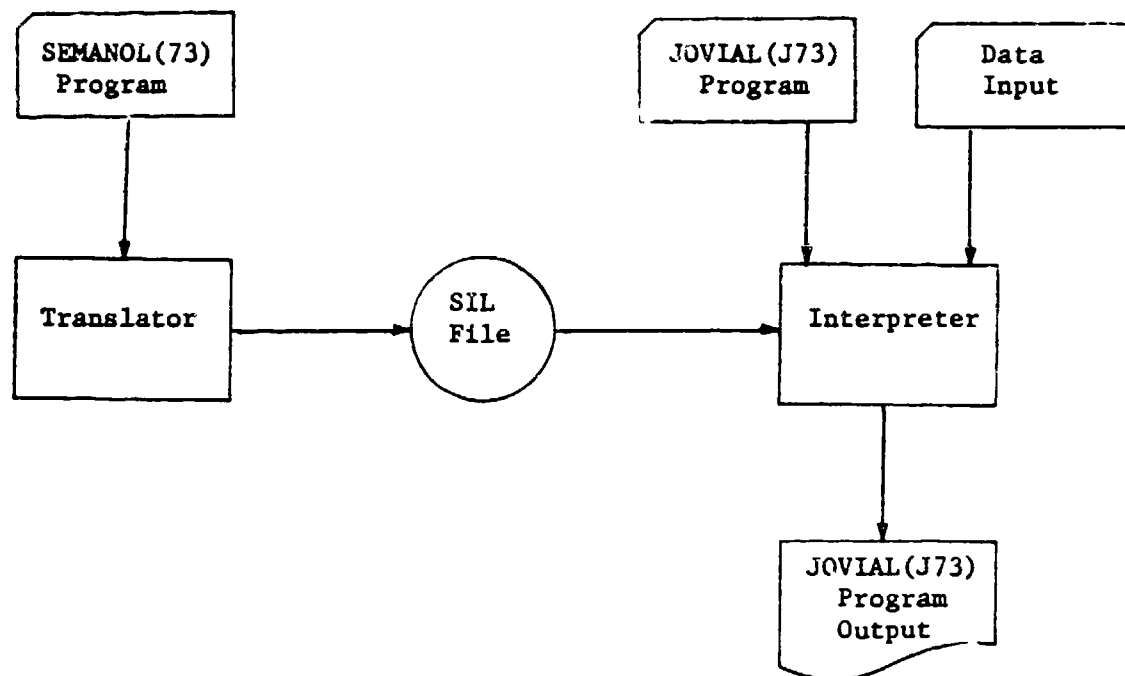
3. Machine Definition

The SEMANOL(73) Interpreter runs on a HIS-6180 computer under control of the Multics operating system. Its core memory requirement depends mainly upon the size of the SEMANOL(73) meta-language program being used. The program is normally operated in a timesharing mode and thus requires that file space be available on the permanent file disk unit. The program does not use magnetic tape, card devices, nor the on-line printer.

4. Program Description

The SEMANOL(73) Interpreter has been implemented through the use of two major subprograms as observed earlier. The first subprogram, the Translator, reads a SEMANOL(73) program describing a programming language and converts it to a form which is readily usable by the next subprogram, the Interpreter. The Interpreter takes the Translator output and uses it to control the execution of programs given to the Interpreter. While the SEMANOL(73) Interpreter is itself programming language independent, to be more specific in the discussion of this report JOVIAL(J73) is always used as the language described by a SEMANOL(73) program.

The relationship of the two subprograms as described is shown in Logic Diagram 1. The SIL file used for communication between the subprograms is described with the Interpreter. It is an alphameric representation of a list of operands and operators produced by processing the SEMANOL(73) program. Note that these subprograms were written in Fortran apart from some PL/I routines which were



Logic Diagram 1. SEMANOL(73) Interpreter

needed for special functions (e.g., doing half word load and stores and double precision integer arithmetic). Each subprogram is described separately in what follows, and indeed they are rather independent due to the minimum interface resulting from the use of the SIL file. Please note that the program listings are richly annotated and contain the full details of program implementation.

The auxiliary program Tnames is optionally run between the Translator and the Interpreter. It is used to translate numerically coded names generated by the Translator back to the corresponding symbolic names from the original source SEMANOL(73) program. This is useful when using the Interpreter trace feature; the trace output will then use the same symbolic def names that the source code contains.

4.1 The Translator

The SEMANOL(73) Translator translates a SEMANOL(73) language source program into a Semanol Interpreter Language (SIL) object program. The Translator uses the recursive-descent method to analyze the syntax of the source program. Recursive descent is a top-down, predictive recognition process employing one recursive procedure or subroutine for each of the rules of source language syntax. This method was chosen because it allows the construction of a modular translator program; changes or additions to the source language syntax are easily accommodated since there is little interdependence among the recognition subroutines.

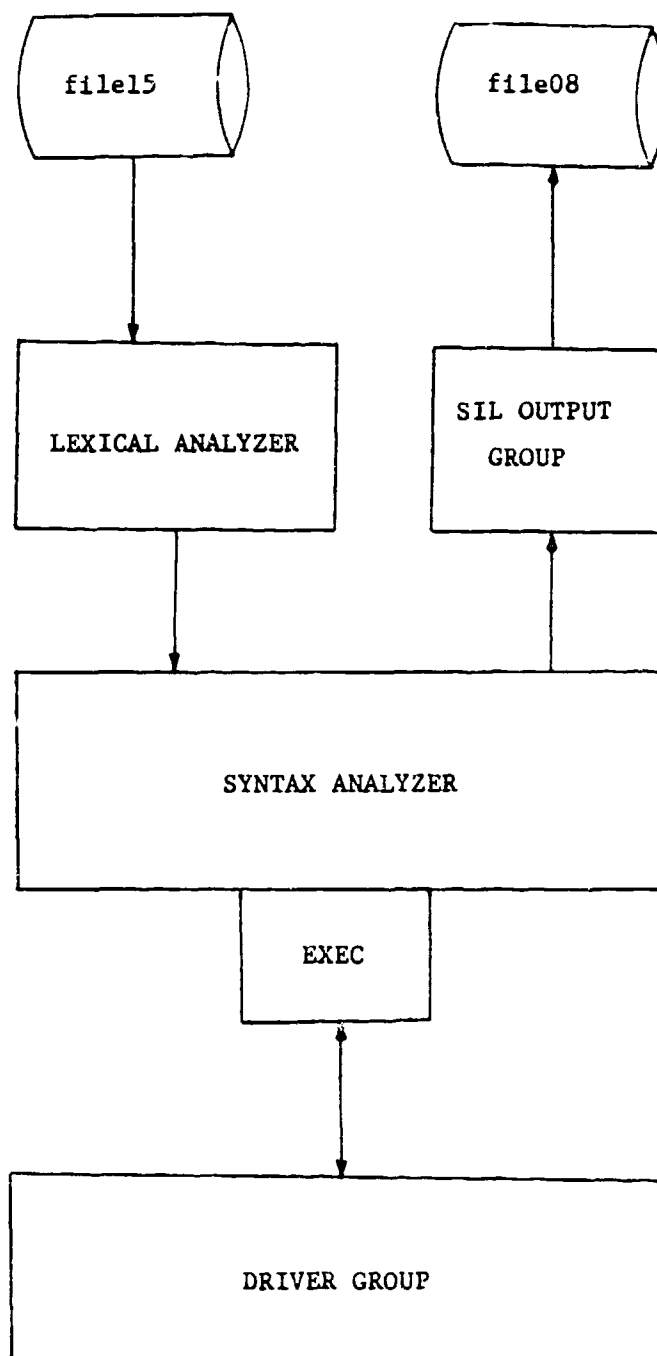
The Translator is divided into two functional parts: lexical analysis and syntactic analysis. In addition to the two main units, there is a driver group to control program execution, a blockdata group for convenient definition of constants, a utility group to handle initialization and cleanup and to provide minor functions such as string-to-integer conversion, and an output group to handle SIL code generation (see Logic Diagram 2).

The driver group consists of a main program and four sub-drivers which correspond to the four divisions of a SEMANOL(73) source program. The sub-drivers recognize their particular heading statement, then instruct the syntax analyzer to begin looking for the constructions expected in that section. Control returns when the syntax analyzer has processed all the statements in the section.

The syntax analyzer consists of recursive subroutines, corresponding to each non-terminal syntax rule of the SEMANOL(73) language. Since the source language of the translator, Fortran, does not have facilities for recursive programming, inter-subroutine communication is handled by a special subroutine, EXEC. Each recursive subroutine has

SOURCE LANGUAGE FILE

SIL OUTPUT FILE



Logic Diagram 2. The Translator Program

three arguments in the calling list:

SUBROUTINE RECUR(FCAL,BCAL,ARG).

FCAL is a forward call code, indicating the next subroutine to be called; BCAL is a return code, composed of the subroutine number and an internal jump number; ARG is a code indicating success or failure of the called subroutine.

One subroutine calls another as shown in the following code example:

```
.  
.   
.   
*CALL STREXP...  
  FCAL=1      (CALL CODE FOR SUBROUTINE STREXP)  
  BCAL=342    (IDENTIFIER FOR THIS SUBROUTINE,34, AND AN INTERNAL  
              JUMP CODE,2)  
  RETURN  
*  
300 CONTINUE  
.  
.  
.
```

The RETURN statement transfers control to EXEC; EXEC stacks the BCAL code and executes a CALL TEXP(FCAL,BCAL,ARG). When there are no more forward call requests, (FCAL=0), EXEC unstacks the previous return code and executes a call to that subroutine. The resulting action is illustrated by the following code, which is the first executable code in each recursive subroutine:

```
*  
* Recursive Entry Routine...  
*  
  IF (FCAL.EQ.34) GO TO 100  
  IF (FCAL.NE.0) TO TO 900 (ERROR CONDITION)  
  J = BCAL-10*(BCAL/10)  
  GO TO (200,300,400),J
```

If FCAL=34, this is a normal forward call, and a jump is taken to the beginning of the code (statement 100); if FCAL=0, this is a return, and the return point is indicated by the units position of BCAL. BCAL was 342 in the previous code example, so J=2 and an internal jump to statement 300

is taken--the desired return point in the previous example.

Recursive descent requires only one global variable--the next symbol element from the source language statement being parsed. As the syntax analyzer proceeds through a statement, it must request the lexical analyzer to place the next symbol into the global variable NEXTSYM; this is accomplished by executing a CALL NEXT statement. Subroutine NEXT is the link between the syntactic and the lexical analyzers.

As the source language statements and statement elements are recognized, the corresponding SIL translation is generated by calls to the SIL output group. These calls are made by the recursive subroutine that recognized a particular element.

The lexical analyzer scans the source language statements and breaks each statement into its smallest recognizable elements. These elements are called tokens or symbols. The lexical analyzer recognizes and types the tokens into the five categories possible:

- Integers
- Names
- Keywords
- Delimiters (+, - etc.)
- Strings.

One source language statement is scanned at a time; the tokens are typed, name-and string-table entries made, then the tokens are placed in a stack. These tokens are made available to the syntax analyzer one at a time, as requested by calls to the interface subroutine NEXT; when the token is exhausted, another source statement is processed by the lexical analyzer.

The output group generates SIL-language output from calling-argument codes passed from the various syntactic analysis subroutines. The SIL output is buffered into 72-character lines, then output on Fortran unit UNITSIL (defined in the blockdata section). Since SIL grammatical order is not the same as in SEMANOL(73), function codes are available which allow the calling-subprogram to cause SIL output to be stacked on different levels, then dumped to UNITSIL.

The blockdata group consists of four blockdata subprograms. BKDATA1 is a set of three tables used for keyword and delimiter recognition. BKDATA2 relates individual ASCII character codes to type codes, grouping the characters as 'lowercase', 'digit', 'uppercase', and so on. BKDATA3 defines constants, such as Fortran unit assignments: UNITIN, UNITOUT, UNITERR, UNITSIL, UNITSCR, UNITSYM, and

UNITNAM; these are, respectively, the SEMANOL(73) source file, file15; the output listing file, file16; the error message file, file16; the SIL output file, file08; a scratch file, file39; the binary symbol table dump file, file37; and the symbolic symbol table dump file, file40. BKDATA4 holds SIL tables. BKDATA1 and BKDATA4 are generated by the programs KTABLD and SILBILD from lists of keywords and SIL tokens.

4.2 The Interpreter

As soon as the Interpreter is loaded, all files are rewound and the INIT subroutine is called. The purpose of INIT is to initialize all of the variables and tables used by the Interpreter. All of these variables and tables are fully described in the internal documentation for the BLKDATA subroutine. Following are described these structures in the order in which they are initialized:

1. The descriptor table is zeroed except for the ITL field of each entry. The free entry list is constructed using this field as a pointer.
2. The string array, ICHAR, is zeroed.
3. The main stack is zeroed.
4. The symbol table bucket array, IBUCK, is zeroed.
5. The type table arrays ITTN, IFORM, JCVLT, JHD, JHDA, JTL are initialized and sorted.
6. Simple global variables are set to their initial values.
7. The variables SMNAM, SMATT, SILPG, PARSE, SYNTAX, IINT, FPNO, STR, STRNM, STNDA, STNDB, SEQ, FCALL, PCALL, SAVE, PTR, PRS, SEQU, LOG, FIX, and UND are initialized to contain their associated type numbers.
8. The variables OP, VAR, SYN, VAL, SIL, and PROC are initialized to contain their associated attribute type numbers.
9. The null sequence, #UNDEFINED, #TRUE, #FALSE, and the null string are pushed onto the stack.
10. Other initial symbol table values are read in from file09.
11. Finally, INIT returns.

While initialization is going on, many of the general utility routines are called. A list of some of these utility routines and what they do follows:

1. The PL/1 functions ITYPE, IHDA, IHD, ITL, ICT, and IPTR are called with a descriptor table or stack index and return the value of the named field at that index.
2. The PL/1 subroutines STYPE, SHDA, SHD, STL, SCT, and SPTR do the inverse; they store a value into the associated field of the descriptor table or stack.
3. LITSTR is used to create a literal string and push its descriptor onto the stack.
4. LKPTP looks up a type name in the type table.
5. LKPN looks up a name in the symbol table.
6. PUSH pushes a value onto the main stack.
7. JSILSM reads a symbol table entry from a file.

These are just a few of the utilities used by the initialization, but they give an idea of the kinds of operations required.

When initialization is complete, the SIL program is read from file08. The JSILSM function is called once for each SIL statement to read. JSILSM must parse each SIL statement and convert it to internal list form.

INTERP is the main Interpreter routine. It calls the operator subroutines as required by the SIL program. Since the flow of control is directed by INTERP, which is in turn directed by the SIL program, an understanding of SIL is essential to an understanding of the Interpreter.

The external syntax of SIL is extremely simple (See Table 1). Internally, the SIL program is stored in a list form; but this fact is independent of its meaning. It will be assumed that the INTERP subroutine works directly on the string format as this assumption elucidates the following discussion.

A SIL program consists of two kinds of statements, syntax statements and routines. An example of a syntax statement is

```
PROGRAM/SYN=(SCAT/OP STMT/SYN (KSTAR/OP STMT/SYN KEND/OP));
```

An example of a routine is

PRINTA/SIL=('A' OIODAT/OP);

The difference between the two is that a syntax statement always has statement-attribute 'SYN' and corresponds to a SEMANOL(73) syntactic #DF. The syntax statements are not executed directly by INTERP, but instead are used by the parsing subroutine JPARSE. To contrast this, routines have statement-attribute 'SIL' or 'PROC'. They correspond to SEMANOL(73) semantic #DF's and SEMANOL(73) commands. They are read directly by INTERP which calls the operator subroutines to execute them.

The syntax statements will be discussed with the parser. Routines can be divided into two types as identified by their statement-attribute. Corresponding to SEMANOL(73) semantic #DF's are routines with statement-attribute 'SIL'. Corresponding to the SEMANOL(73) program in the #CONTROL-COMMANDS section is the routine with statement-attribute 'PROC'. The subroutine INTERP treats all of these routines the same once execution begins. The difference is that they contain different varieties of SIL code corresponding to different types of SEMANOL(73) statements.

The elements within a SIL statement list are normally processed from left to right. In fact, within a list, the element string is like a reverse Polish string of operators and operands. Operands and results of operations are kept on a stack. The actions taken when each kind of element is encountered are summarized below:

1. If a (sub) list is encountered, the Interpreter saves (on the stack) its current position and begins processing the elements of the sub-list.
2. If a <name><'/'><statement-attribute> or <name><'/'><value-attribute> is encountered, a pointer to the symbol table entry for the given element is pushed onto the stack. Note that each <name><'/'><attribute> has its own symbol table entry.
3. If a constant is encountered, its value is pushed onto the stack.
4. If a <name><'/'><operator-attribute> is encountered, the operator with the given name is executed.
5. If the end of a (sub) list is encountered, the Interpreter continues execution at the element after the last one saved as in step 1.

At this point, consider case 1 above. The INTERP subroutine handles each operator by either calling a subroutine whose

name is the same as the operator subroutine name or by jumping to the appropriate code in its own body. The operators which require operands take them from the top of the stack. They often replace their operands with a result.

The operands for functional operators are descriptors for, or pointers to, the various SIL data types. The data types used are:

1. UND - #UNDEFINED
2. FPNO - A double floating point number, high order bits
3. IINT - An integer
4. PRS - A parse tree
5. SEQ - A finite sequence
6. STR - A string
7. FIX - A floating point number, low order bits
8. LOG - #TRUE or #FALSE.

The stack contains pointers to symbol table entries and it also contains special entries marking function calls, sublist calls, and for saving old values of parameters in recursion.

Consider the following (sub) list:

(1 2 A/VAR CVS/OP STLFT/OP STRIT/OP A/VAR ASVAR/OP)

Suppose the Interpreter encountered this list as an element of another list. It would first mark its place in the other list by pushing a PCALL node on the stack and then begin this list by pushing the integers 1 and then 2 onto the stack. When it encountered A/VAR it would push a pointer to the symbol table entry for A/VAR onto the stack. But then CVS/OP (convert to string operator) would go to the A/VAR symbol table entry, get the value stored there, convert it to a string, and push the string descriptor onto the stack in place of the pointer to A/VAR. The top of the stack (at right) would now contain

...PCALL,1,2,'ABC'

given that A/VAR had a value of 'ABC'. The Interpreter would then call the STLFT subroutine which would replace 2 and 'ABC' with 'AB', implementing the SEMANOL(73) #LEFT 2 #CHARACTERS-OF A. Next, the STRIT subroutine would be called to replace 1 and 'AB' with 'B', implementing the

SEMANOL(73) #RIGHT 1 #CHARACTERS-OF (#LEFT 2 #CHARACTERS-OF A). The top of the stack would now contain

...PCALL, 'B'

Next, another pointer to the A/VAR symbol table entry would be pushed on the stack and the ASVAR subroutine would be called to store 'B' at the A/VAR symbol table entry location. ASVAR deletes its arguments without leaving anything on the stack, so there would now be nothing above the PCALL node on the stack. Finally, the right parenthesis (end of sub-list) would be encountered and the PCALL node would be removed from the stack. Control would return to the previous list.

Not all operators pass control to the next sublist element in the normal way. The operators which start with K (e.g., KTRUE/OP, KFALSE/OP) can either succeed or fail. If one of these operators succeeds, control proceeds as usual. But if one fails, a premature end to the sublist takes place. It is as if the sublist end had been encountered and control returns to the calling list. An example of how this feature is used is as follows:

((B/VAR CVS/OP 'A' PEQW/OP KTRUE/OP MERR/OP) MSTOP/OP)

Suppose control has come to the B/VAR element. A pointer to the B/VAR symbol table entry is pushed onto the stack. The CVS subroutine converts this pointer to the string value of the variable B. 'A' is then pushed onto the stack. Now the PEQW subroutine compares the top two stack entries (assuming they are strings). If they are identical it replaces them with #TRUE and if not it replaces them with #FALSE. (This implements the SEMANOL(73) arg1 #EQW arg2.) Now, the KTRUE function is called. If the top stack entry is #TRUE it succeeds and control next goes to the MERR subroutine. Otherwise, KTRUE fails and control goes to the MSTOP subroutine.

A number of other operators can interrupt normal left to right processing within a sublist. They include the following:

CALL/OP - CALL is used to implement a SEMANOL(73) #DF call. When it is encountered in a routine list, the top of the stack contains (starting at top) a pointer to the symbol table entry containing the routine for the #DF to be called, an integer whose value is the number of arguments being passed to the #DF, and finally, the argument values themselves.

RET/OP - RET implements a SEMANOL(73) non-procedural

#DF return. When it is encountered, control returns to the point at which the last CALL/OP was executed. The value returned is the one stored at dname/VAR where dname is the name of the #DF which is returning.

- LOOP/OP - LOOP normally occurs at the end of a sublist. It causes control to resume at the beginning of that sublist. As its name implies, it is used to implement loops.
- ENDIF/OP - ENDF also normally occurs at the end of a sublist. It is somewhat like a normal right parenthesis at the end of a sublist except that control returns up 2 sublist levels instead of one. This is used to implement #DF's with cases.

Now, an example of a SEMANOL(73) semantic #DF is given, showing how it is represented as an SIL statement, and then following INTERP through its interpretation. This will illustrate the #DF calling mechanism of the Interpreter. The SEMANOL(73) #DF follows:

```
#DF LONGER(StringA,StringB)
```

```
=> StringA #IF #LENGTH(StringA) >= #LENGTH(StringB);
=> StringB #OTHERWISE #.
```

This #DF returns the longer of its two string arguments. It translates into the SIL statement

```
LONGER/SIL=(2 IPARAM/OP LONGER/VAR LOCAL/OP
StringA/VAR PARAM/OP StringB/VAR PARAM/OP FCALL1/OP
((StringA/VAR CVS/OP ISLEN/OP StringB/VAR CVS/OP ISLEN/OP
PLT/OP LNOT/OP KTRUE/OP StringA/VAR LONGER/VAR ASVAR/OP ENDF/OP)
(StringB/VAR LONGER/VAR ASVAR/OP ENDF/OP))
RET/OP);
```

Suppose for the illustration that the arguments are StringA='AB' and StringB='ABC' when the #DF is called. The Interpreter finds itself at the 2 in the above code. The first two lines are similar to those that head the SIL code for any SIL #DF. They indicate to the Interpreter the number and names of the parameters and tell it to save on the stack the old values of these and the variable with the same name as the #DF. They also mark the stack so the Interpreter will return correctly. The next two lines of code indicate that the Interpreter must calculate and compare the lengths of the two strings. Since StringA is shorter, the KTRUE/OP on the second of the two lines will

fail. Control will skip over the rest of the sublist to the line starting with '(STRINGB/VAR'. This line assigns STRINGB (the correct result) to the variable with the same name as the #DF because that is the way #DF values are returned. The ENDIF/OP then skips the Interpreter out to the RET/OP which causes a return to the point which called the whole #DF.

The following SIL code calls the parser to parse the string at S/VAR using as root production the syntax #DF with left-hand-side PRODUCTION:

S/VAR CVS/OP PRODUCTION/SYN STPRS/OP

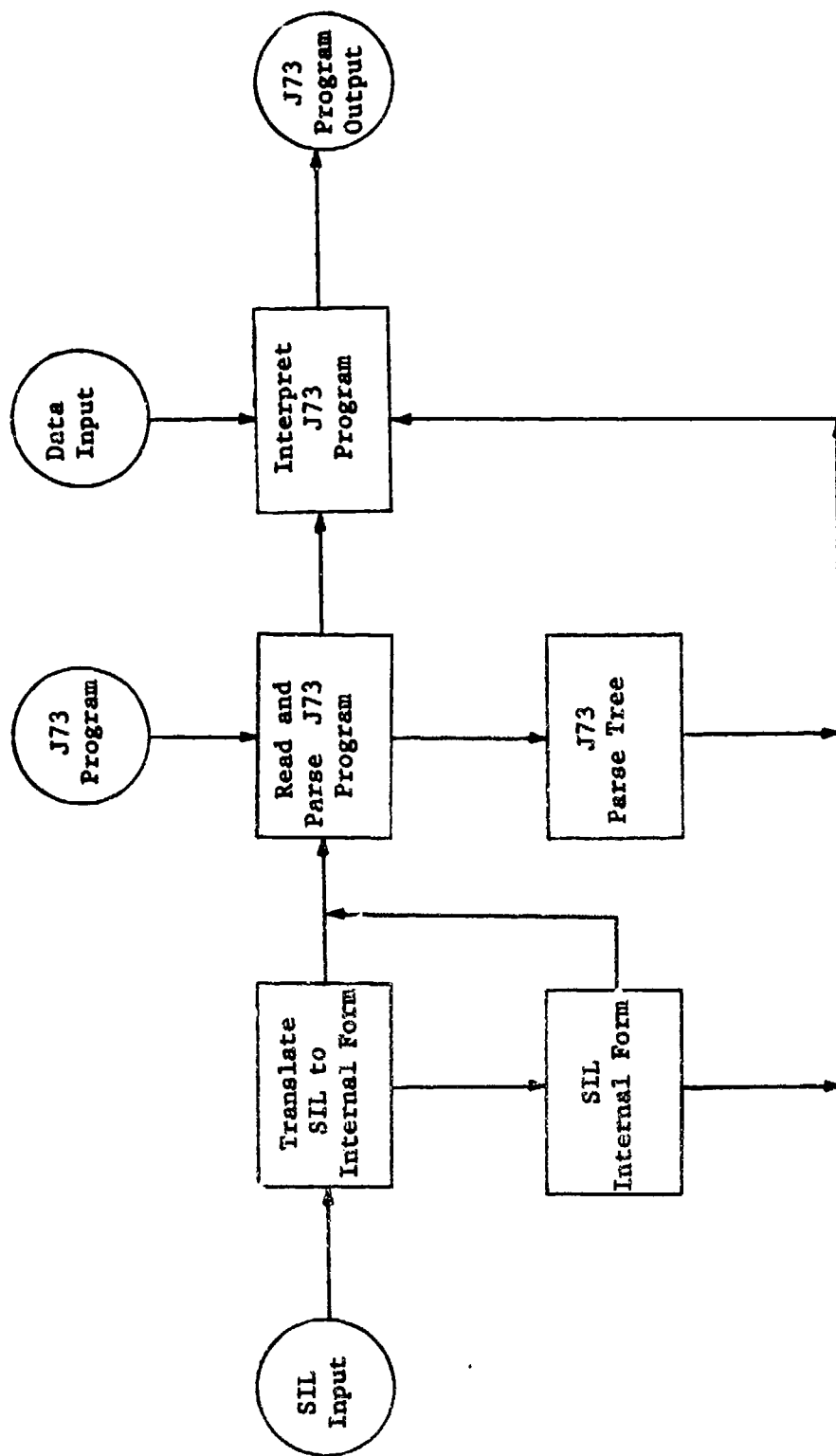
The context-free grammar used by the parser is written using SEMANOL(73) syntactic #DF's. It is translated into SIL syntax statements as previously stated. One SIL syntax statement corresponds to each SEMANOL(73) syntactic #DF. Note that the set of legal SIL syntax-statements is a subset of the set of all SIL statements (See Table 2). The parser uses the Jay Earley parsing algorithm, modified for SEMANOL(73), to compute a parse tree. The algorithm is documented in the listing and in several papers by Earley, so is not described here.

This concludes the discussion of INTERP which, indeed, passes control directly or indirectly to almost every other Interpreter subroutine at some time or other. The only thing that has not yet been mentioned is how INTERP halts. This can happen in one of two ways:

1. INTERP encounters MSTOP/OP in the SIL program control stream.
2. One of the operator subroutines detects an error of some kind, either in the JOVIAL(J73) program, the SIL program, or the data. In this case an appropriate error message is printed on the output file.

When either of these things happens, the Interpreter either halts completely or it re-initializes itself to execute another SEMANOL(73) program (if one exists).

A simplified representation of the Interpreter is given in Logic Diagram 3.



Logic Diagram 3. The Interpreter

Table 1. SIL Syntax

```
#DF statement => <statement-name><'/'><statement-attribute><#GAP>
<'='><#GAP><list><#GAP><'> #.

#DF statement-name => name #U ['#PROC'] #.

#DF statement-attribute => ['#SYN', 'SIL', 'PROC'] #.

#DF list => <'('><#GAP><element><%<<#GAP><element>>>
<#GAP><')> #.

#DF name => <'#ALPHABET'>
<%<<#ASCII> #S- [' ', '/' ]>> #.

#DF element => list
=> <name><'/'><statement-attribute>
=> <name><'/'><operator-attribute>
=> <name><'/'><value-attribute>
=> constant #.

#DF operator-attribute => ['OP'] #.

#DF value-attribute => ['VAL', 'VAR'] #.

#DF constant => <any legal SEMANOL(73) string constant>
=> <'#B'><any legal SEMANOL(73) bit-string constant>
=> <'#R'><any legal SEMANOL(73) floating-constant>
=> <'#I'><any legal SEMANOL(73) integer-constant>
=> <'#TRUE'>
=> <'#FALSE'>
=> <'#UNDEFINED'>
=> <'#NULLSQ'>
=> <#DIGIT><%<<#DIGIT>> #.
```

Table 2. SIL Syntax for Syntactic Statements

```
#DF syntax-statement => <statement-name> <'/'><syntax-
statement-attribute><#GAP> <'='><#GAP><syntax-list>
<#GAP><';'> #.

#DF statement-name => name #.

#DF syntax-statement-attribute => ['SYN'] #.

#DF syntax-list => case
=> scat #.

#DF name =><#ALPHABET>
<% <<#ASCII> #S- [' ', '/' ]>> #.

#DF case => <'('> <#GAP> <'CASE/OP'> <#GAP>
<%1< <cat> <#GAP>>> <')'> #.

#DF scat => <'('> <#GAP><'SCAT/OP'><#GAP>
<%1< <prim> <#GAP>>> <')'> #.

#DF cat => <'('><#GAP><%1<<prim><#GAP>>><')'> #.

#DF prim => set
=> union
=> setmin
=> kstar
=> kstar1
=> scanop
=> nterm
=> strlit #.

#DF set => <'('><#GAP><'SET/OP'><#GAP>
<%1<<strlit><#GAP><'SETEND/OP'><#GAP>>><')'> #.

#DF union => <'('><#GAP><'UNION/OP'><#GAP>
<%1<<alt><#GAP>>><')'> #.

#DF alt => <'('><#GAP><%1<<prim><#GAP>>>
<'UNEND/OP'><#GAP><')'> #.

#DF setmin => <'('><#GAP><'SETMIN/OP'><#GAP>
<%1<<prim><#GAP>>><'SMEND/OP'><#GAP><'SMENDA/OP'>
<#GAP><%1<<strlit><#GAP>>> <')'> #.

#DF kstar => <'('><#GAP><'KSTAR/OP'><#GAP>
<%1<<prim><#GAP>>><'KEND/OP'><#GAP><')'> #.

#DF kstar1 => #<'('><#GAP><'KSTAR1/OP'><#GAP>
<%1<<prim>#GAP>>><'KEND/OP'><#GAP><')'>#.
```

```
#DF scanop => [ 'DIGIT/OP', 'SPACE/OP', 'ALPHA/OP',  
  'ASCII/OP', 'GAP/OP', 'NIL/OP', 'CAP/OP', 'DNUM/OP',  
  'LCASE/UP', 'NATNO/OP' ] #.
```

```
#DF nterm => <name><`/SYN`> #.
```

```
#DF strlit => any SEMANOL(73) string literal
```

4.3 Tnames

After initializing itself, Tnames reads file20, a symbolic dump of the Translator symbol table, into its own internal table. It then reads lines one at a time from file21, the Translator SIL file. A line is skipped if it is blank and Tnames goes to the next line. If the line is not blank, Tnames searches it for Translator names of the form NDDDD (where D is a digit). Each name found is looked up in the Tnames internal table and the associated Semanol source name is substituted for the Translator name. As each line is completed, it is written to file22, and then Tnames goes to the next line. Tnames STOPS when all of file21 has been read.

5. Logic Diagrams

The Logic Diagrams were given earlier. There is an overall logic diagram and then one for each of the two main programs which constitute the Semanol Interpreter.

6. Inputs

Program inputs are discussed separately for the programs.

6.1 Translator Input

There is only one input file to the translator and that is Fortran file15. This is an ASCII file which contains the SEMANOL(73) source language program.

6.2 Interpreter Input

Three input files, file07, file08, and file09 are required to run the Interpreter. They are ASCII files like all other Multics text files.

file09 contains data used to initialize the symbol table entries with attribute OP. For each operator used by the Interpreter, a line on this file must contain

opname/OP = number;

where <opname>

is the operator name and

<number>

is its computed goto index in the INTERP subroutine. The file ends with a line with four sharps.

file08 contains the SIL version of the SEMANOL(73) programs to be run. There are several possible cases:

1. If only one SEMANOL(73) program is on the file and it is to be run once for each JOVIAL(J73) program on file07 with complete system initialization only at run initiation, then the SEMANOL(73) program is followed by a single line with three sharps at the beginning.
2. If only one SEMANOL(73) program is on the file and it is to be run once for each JOVIAL(J73) program on file07 with the complete system initialization and re-reading of the SEMANOL(73) program between each run, then the SEMANOL(73) program is followed by two lines with four sharps and a blank card.
3. If many SEMANOL(73) programs are on the file, they are each followed by a line with four sharps. Another line with four sharps follows all other cards. In this case, complete system initialization occurs after each run.

file07 contains the JOVIAL(J73) programs to be run along with their input data. As many programs can be run as desired. The JOVIAL(J73) program for each run is followed by a four sharp card (sharps in columns 1 through 4). The data for that run then comes followed by a line with ###* and then a three sharp card.

6.3 Tnames Input

The Tnames program has two input files, file20 and file21. They are ASCII Multics text files.

file20 contains the symbol table information that the Translator left on file40. (Note that file20 can be a link to file40 after running the Translator.) Each line of the file is of the form

DDDD NAME

where DDDD is a four digit code indicating that the Translator name NDDDD corresponds to the Semanol source name NAME.

file 21 contains the SIL code as output by the Translator onto file08. This SIL code does not, obviously, use the Semanol source names.

7. Output

The program output files are discussed individually for each program.

7.1 Translator Outputs

All output files are in ASCII format except file37. The following Fortran unit files are used:

1. file08 contains the generated SIL output.
2. file16 contains the error message listing.
3. file39 is a scratch file.
4. file37 is a binary dump of the translator symbol table.
5. file40 is a symbolic dump of the symbol table names. See Section 6.3 for a description of this file which is used only by Tnames.

7.2 Interpreter Outputs

The output file is file06. This file is the terminal. All JOVIAL(J73) output goes to this file. Other things which go to this file are:

1. Any error messages output by Fortran.
2. Any error messages output by the Interpreter.
3. Messages indicating when the Interpreter garbage collector (GGC) and string compactor (SCOMP) are called and when they return.
4. #DF tracing messages when they are enabled.
5. A message "MSTOP CALLED" after each SEMANOL(73) program on input file08 is run.

7.3 Tnames Output

The Tnames program has one ASCII Multics text output file, file22. file22 contains the SIL code that was on the input file21 except that all blank lines are deleted and Semanol source code names have been substituted for Translator names of the form NDDD.

8. Program Setup

The Translator and Interpreter are run in MULTICS timesharing mode. In addition to the program segments, file segments must be provided for the Translator as follows:

file15	Semanol program source code
file16	Translator listing and error message file
file08	SIL code output file
file39	scratch file
file37	binary symbol table dump file
file40	symbolic symbol table dump file.

The Fortran unit assignments are made in the blockdata subprogram BKDATA3. They are:

UNITIN	- file15
UNITOUT	- file16
UNITSIL	- file08
UNITERR	- file16
UNITSCR	- file39
UNITSYM	- file37
UNITNAM	- file40.

Should it be necessary to use different unit assignments, only BKDATA3 need be changed.

The six file segments must be present in directory Blum (or proper links set up to other directories) prior to running the Translator. The SIL output file, file08, will be used by the Interpreter portion or Tnames for further processing.

For the Interpreter, files file07 (JOVIAL(J73) program and data in form described under Interpreter Input) and file09 (initialization) must be set up in directory ERAnderson. Interpreter binaries must be in directory ERAnderson. The system is then ready to run.

9. Operator Instructions

We assume that the programs are set up in the directories as stated above.

```
login Blum
ekb
sem
logout -hd
login ERAnderson
era
link <Blum>file08
main
```

The above procedure will first cause the Translator to be executed and then the Interpreter. (Tnames will not be run.) The Interpreter output will appear on the terminal.

```
login Blum
ekb
sem
logout -hd
login ERAnderson
era
link <Blum>file08 file21
link <Blum>file40 file20
tnames
new_proc
link file22 file08
main
```

The above procedure will first cause the Translator to be executed, then Tnames, and finally, the Interpreter. The Interpreter output will appear on the terminal. This procedure is used if tracing is to be done during interpretation so that Semanol source symbolic names will be used.

An exec-com procedure <Belz>buildj73, is available to build the SEMANOL(73) description of JOVIAL(J73) on file15. The input to <Belz>buildj73 is the set of files <Belz>j73.a, <Belz>j73.b,...,<Belz>j73.l; these files are in runoff source form. The output is on file15. in proper form for input to the translator.

To build file15 prior to translating the JOVIAL(J73) description, use the following commands:

```
login Blum
ekb
ec <Belz>buildj73
```

Another exec-com procedure <Belz>buildj73.list builds a publication listing of the SEMANOL(73) description of

JOVIAL(J73) on file jovial.j73.sem; it uses the same input files as <Belz>buildj73 uses.

To build a listing of the JOVIAL(J73) description, login under any account, and use the command

ec <Belz>buildj73.list

These programs run under the standard Multics system and thus require no special instructions to the computer operator. These programs use no magnetic tape or other removable storage media.